1

Clases y Objetos

Clases y Objetos

Atributos, métodos, constructores

Sobrecarga de métodos y de operadores

Variables y métodos de clase

Variables y métodos de instancia

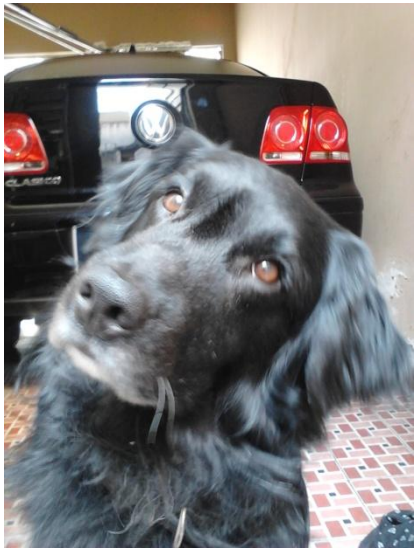
Rafael Aguilar García – ESCOM - IPN

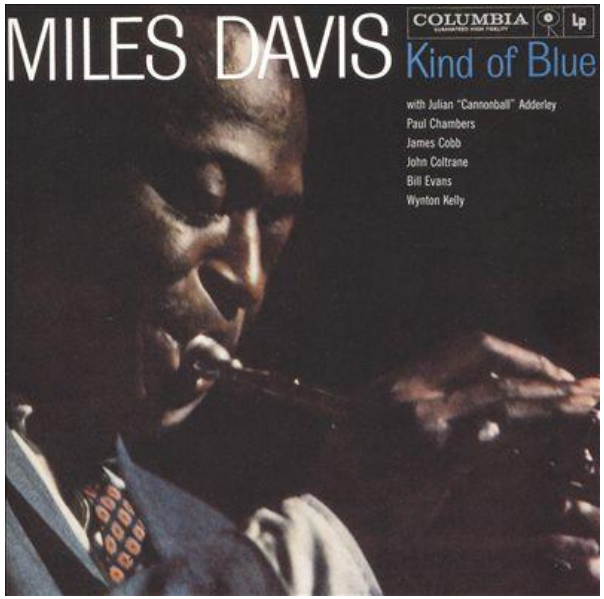
Objetos

- En el mundo real un objeto es una entidad, un nombre, o cualquier cosa que tenga una identidad propia.

Objetos – Categorías de Objetos

- Cosas tangibles:





Objetos – Categorías de Objetos

- Cosas intangibles:

Internet

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD
  XHTML 1.0 Transitional//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/
  xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/
  xhtml">
5     <head>
6         <meta http-equiv="Content-
  Type" content=
7             "text/html; charset=us-
  ascii" />
8         <script type="text/
  javascript">
9             function reDo() {top.
  location.reload();}
10            if (navigator.appName ==
  'Netscape') {top.onresize = reDo;}
11            dom=document.
  getElementById;
12        </script>
13    </head>
14    <body>
15    </body>
16 </html>
```

Programa

Vuelo

Objetos – Categorías de Objetos

- Organizaciones:

ESCOM

Fábrica

Hospital

Objetos – Categorías de Objetos

- Roles:

Gerente

Profesor
Alumna

Obrero

Objetos

- Los objetos tienen:
  - Atributos
  - Comportamiento

Objetos – Ejemplo: Robot

- Atributos

Nombre: PaPeRo

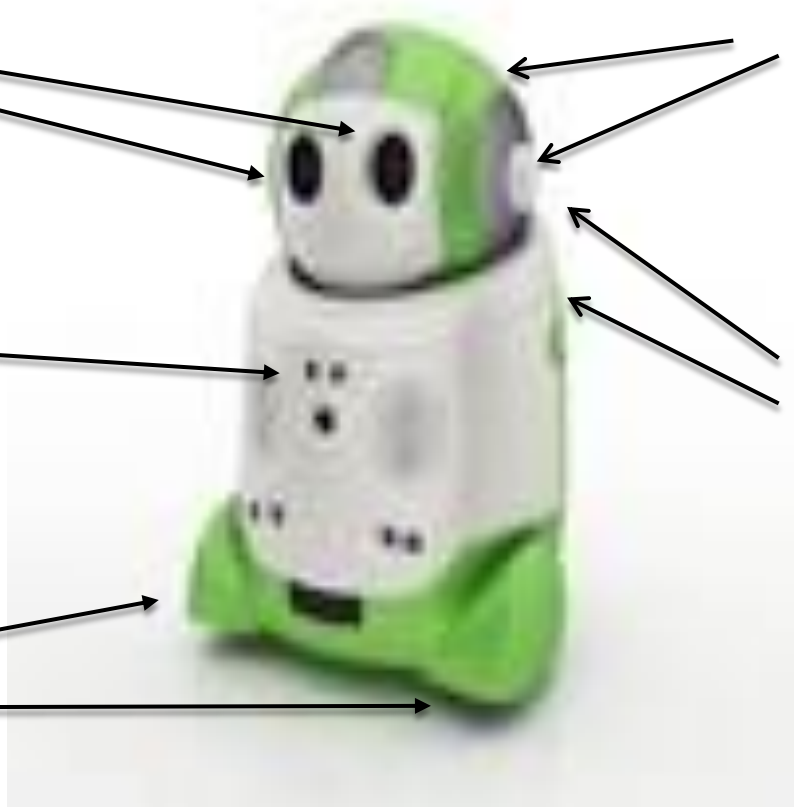
Dos cámaras para reconocimiento de rostro

Dos micrófonos para reconocimiento de voz

Sistema de ultrasonido para localizar objetos

Sensores para percibir palmadas o golpes

Motores



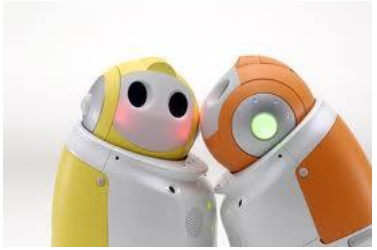
Objetos

- Comportamiento

Avanzar y esquivar objetos

Dar mensajes a la persona apropiada

Saludar



Recargar batería

Reconocer voz

Objetos

- Ejercicio. Menciona los atributos y comportamiento de los siguientes objetos
  - Empleado
  - Puesto de trabajo
  - Empresa
  - Estudiante
  - Materia
  - Profesor
  - Escuela

Clase

- Una **clase** es un prototipo (plantilla) que contiene una colección de **datos** (atributos) y un conjunto de **métodos** (comportamiento) que manipulan los datos o que interactúan con objetos relacionados.
- Una clase es como una fábrica que podemos usar para **instanciar** (crear) múltiples objetos individuales
- Un **objeto**, por lo tanto, es una **instancia de la clase**.
- A los **datos** y a los **métodos** se les conoce como **miembros de la clase**.

Clase

Clase Planeta

Tierra

Marte

Objetos de la clase Planeta

Saturno

Urano

Clase

Clase Automóvil

Ferrari

Audi

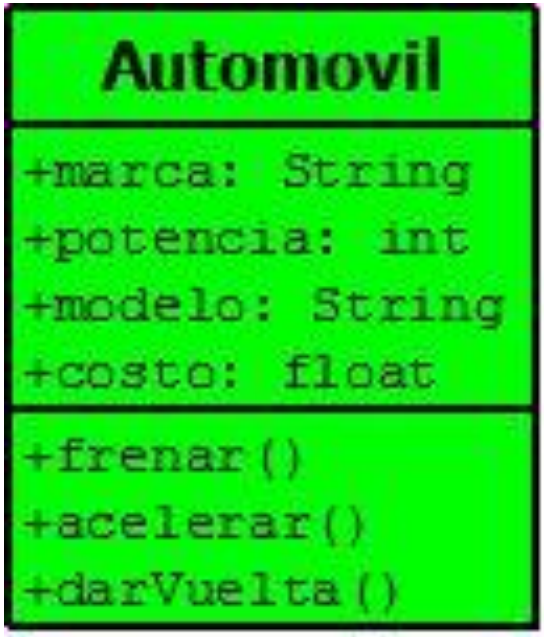
Objetos de la clase Automóvil

Lamborghini

Mercedez Benz

Clase

- Mediante diagramas **UML** (Unified Modeling Language) podemos modelar de forma gráfica una clase:



Nombre de la clase

Atributos de la clase

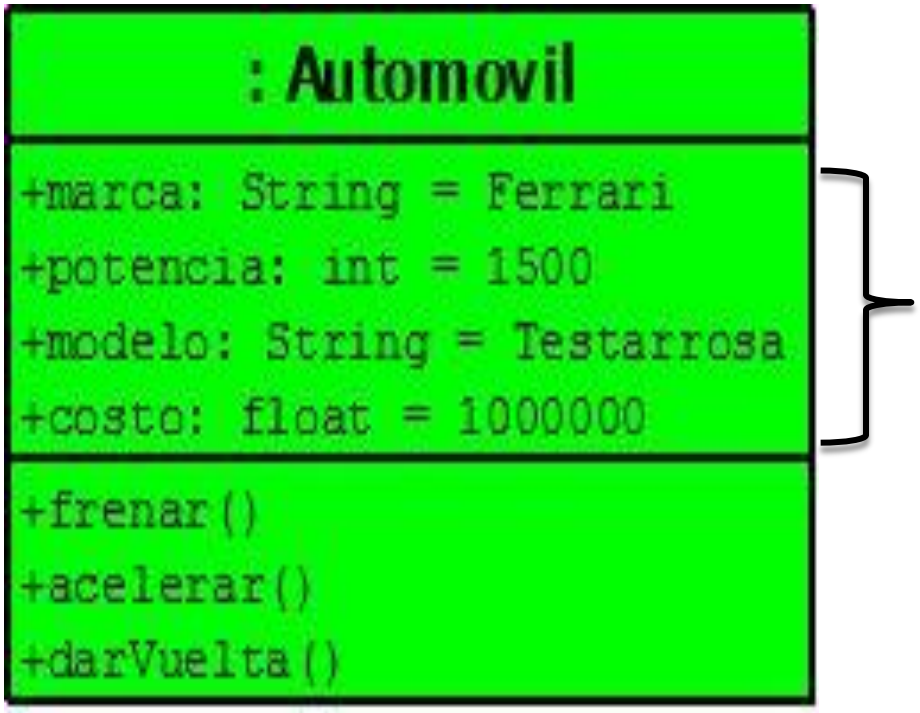
Métodos de la clase

+ = Acceso público

Diagrama UML de la clase automóvil

Clase

- Diagrama UML de un objeto de la clase Automóvil



Los valores de los atributos determinan el estado del objeto de la clase Automóvil

Clases y Objetos

- En la POO nuestros objetos del mundo real migran de forma natural a código orientado a objetos.
- Los objetos se convierten en módulos independientes, y cada objeto sabe qué operaciones (métodos) se aplican a los datos (atributos).

Constructores

- Un constructor es un conjunto de instrucciones utilizadas para inicializar la instancia de una clase.
- Un constructor siempre tiene el mismo nombre de la clase
- Es posible definir uno o más constructores en una misma clase.
- También es posible no definir ningún constructor, en este caso tendremos el constructor por omisión proporcionado por el lenguaje de programación utilizado.

Constructores

- La clase **Complejo** modela un número complejo y la manejaremos para ilustrar el uso de los constructores .

Complejo
-re: int -im: int
+Complejo() +Complejo(r:int,i:int) +Complejo(var:Complejo) +sumar(var:Comlplejo,): void +sumar(var1:Complejo,var2:Complejo): void +restar(var:Complejo): void +restar(var1:Complejo,var2:Complejo): void +imprimir(): void

Constructores

- La clase **Complejo** tiene tres constructores. Esto significa que tenemos tres formas de inicializar una instancia de la clase **Complejo**:

Complejo( )

- Este constructor no tiene argumentos, aquí tanto la parte real como la imaginaria se inicializan al valor de uno.

Complejo(r:int, i:int)

- Este constructor tiene dos argumentos (**r**, **i**) de tipo entero. El valor de **r** se asigna a la parte real y el valor de **i** a la parte imaginaria de la nueva instancia.

Constructores

Complejo(var: Complejo)

- Este constructor tiene un argumento (**var**) de tipo **Complejo**. Este constructor asignará la parte real del objeto **var** al campo real de nuestra nueva instancia, así como la parte imaginaria del objeto **var** al campo de la parte imaginaria de nuestra nueva instancia. A este constructor se le conoce como constructor por copia.

Constructores y métodos

- A continuación se ilustra el siguiente código en el lenguaje de programación Java:
- Clase **Complejo** con sus atributos **re** para la parte real e **im** para la parte imaginaria; los tres constructores ya descritos; y sus métodos **sumar**, **restar**, e **imprimir**. (Archivo: Complejo.java)
- Clase **UsoComplejo** que ilustra la forma de usar la clase **Complejo**, creando instancias y usando los método **sumar**, **restar**, e **imprimir**. (Archivo: UsoComplejo.java)

```
public class Complejo {
    private int re;  // Parte real; solo la clase tiene acceso
    private int im; // Parte imaginaria;  "      "        "
   //Primer constructor
   Complejo( ){
        re = 1;
        im = 1;
    }
    //Segundo constructor
   Complejo(int r, int i){
        re = r;
        im = i;
    }
 //Tercer constructor
   Complejo(Complejo var){
        re = var.re; // var.re es la parte real del objeto var
        im = var.ri; // var.im es la parte imaginaria del
        }                    // objeto var
```

```
//Continuación de la clase Complejo
// Definición de los métodos de suma
   void sumar(Complejo var ){
        re = re + var.re;
        im = im + var.im;
    }
   void sumar(Complejo var1, Complejo var2 ){
        re = var1.re + var2.re;
        im = var1.im + var2.im;
    }
// Definición de los métodos de resta
   void restar(Complejo var ){
        re = re - var.re;
        im = im - var.im;
    }
```

```
//Continuación de la clase Complejo
// Continuación de la definición de los método de resta
void restar(Complejo var1, Complejo var2 ){
        re = var1.re - var2.re;
        im = var1.im - var2.im;
    }
// Definición del método imprimir que despliega los
// valores de la parte real e imaginaria de un objeto
   void imprimir( ){
        System.out.println(" re =  " + re + "im = " + im);
    }
} // Fin de la clase Complejo
```

Archivo: Complejo.java

```
public class UsoComplejo {
    public static void main(String [ ] args){
    //Creamos tres instancias de la clase Complejo
    // usando los tres constructores.
    // Las variables A, B, y C son referencias a un
    // Complejo.
    // El operador new genera la memoria para la
    // instancia de la clase Complejo.

    // Primer constructor sin argumentos
    Complejo A = new  Complejo( );
    // Segundo constructor con dos argumentos enteros
    Complejo B = new  Complejo(5, 8);
    // Tercer constructor con argumento tipo Complejo
    // C es inicializado con los valores de B
    Complejo C = new  Complejo(B);
```

```
    // Uso de los métodos.
    // Imprimimos el contenido de cada objeto mediante
    // el método imprimir( )
      A.imprimir( );
      B.imprimir( );
      C.imprimir( );
     // Ahora sumamos A con B, esto es:  A = A + B
      A.sumar(B);
     // Comprobamos la suma con el método imprimir( )
     A.imprimir( );
     // Ahora sumamos A con B y asignamos el resultado al
     // objeto C, esto es:  C = A + B
     C.sumar(A, B);
     // Comprobamos la suma con el método imprimir( )
     C.imprimir( );
   } // Fin del método main( )
} // Fin de la clase UsoComplejo
```

Archivo: UsoComplejo. java

Ejercicios

- Modificar y/o agregar nuevos métodos a la clase **Complejo** de forma que soporte el siguiente código:
- **A.igualQue(B);**
  - Compara el **Complejo** A con el B, y si son iguales entonces devuelve el valor **true**, en caso contrario devuelve el valor **false**. Probar este método en una sentencia **if**.
- **A = B.sumar(C);**
  - La suma del **Complejo** B con el C es asignada al **Complejo** A.

Sobrecarga de métodos

- Un método puede tener más de una definición dentro de una misma clase; la forma de identificar cuál de los métodos es el referido será con base en el número y tipo de sus argumentos. A esto se le conoce como **Sobrecarga de métodos**.
- En la clase **Complejo** tenemos sobrecarga de métodos en los métodos **sumar** y **restar**. Cada uno tiene dos definiciones; la forma de distinguir cuál es el método invocado será de acuerdo con el número y tipo de los argumentos.

Sobrecarga de operadores

- Supongamos que tuviéramos el siguiente código:

```
Complejo A = new  Complejo( );
Complejo B = new  Complejo(5, 8);
Complejo C = new  Complejo(B);

A = B + C;
C = B – A;
```

Sobrecarga de operadores

- En las líneas 5 y 6, los operadores binarios + y – tienen como operandos objetos del tipo de dato **Complejo**. A esto se le denomina **Sobrecarga de operadores**.
- En la sobrecarga de operadores tenemos que los operadores cumplen una doble funcionalidad: soportan operaciones con operandos de tipos de datos primitivos del lenguaje, y además soportan operaciones con operandos de tipos de datos definidos por el usuario (clases), siempre y cuando éstos estén definidos.

Sobrecarga de operadores

- La sobrecarga de operadores sí está permitida en el lenguaje de programación C++, mediante la función **operator.**
- La siguiente diapositiva ilustra el uso de la función **operator** para implementar la sobrecarga del operador de adición (**+**) en nuestra clase **Complejo** (abreviada), ahora en código fuente de C++

```
class Complejo{
    private:
        int re;
        int im;
    public:
    Complejo(int r, int i )  //Constructor
            : re(r), im(i)
            { };
  Complejo operator + (const Complejo& otro);
};
//Definición de la función operator+
Complejo Complejo::operator+( const Complejo& otro)
  {
    int aux_real = re + otro.re;
    int aux_im = im + otro.im;
    return Complejo(aux_real, aux_im);
}
```

Sobrecarga de operadores

- El lenguaje de programación Java no soporta la sobrecarga de métodos por lo que se deberá trabajar con un estilo de programación como se hizo en la clase **UsoComplejo**:
- **A.sumar(B)** // **Equivalente a: A = A +B**
- Java cuenta con el operador **+** sobrecargado para concatenación de objetos tipo **String**. Pero está predefinido, el programador no puede redefinirlo.

Variables y métodos de clase

- Las variables y métodos de clase pertenecen exclusivamente a la clase; las instancias de la clase no tendrán copia de ellas.
- La palabra reservada **static** declara a una variable de clase o a un método de clase.
- Una variable o un método de clase se comporta como una variable o método global, accesible para todas las instancias de la misma clase y de otras.
- La palabra reservada **static** es soportada tanto en Java como en C++.

Variables y métodos de clase

- En el siguiente código en Java se ilustra el uso de variables y métodos de clase. El programa usa una variable de clase (**contador**) para llevar la cuenta de las instancias de la clase **Empleado**; y un método de clase (**reportarEmpleados**) para reportar la información del número de empleados en cualquier momento.

```
public class Empleado{
    //Variables de instancia
    private String nombre;
    private String  RFC;
    //Variable de clase
    public static int contador=0;
    //Constructor
    Empleado(String n, String rfc){
      nombre = n;
      RFC = rfc;
      contador++;
      }
      //Método de clase
     public static void reportarEmpleados( ){
  System.out.println("El numero de empleados es: " + contador );
     }
```

```
// Continuación de la clase Empleado
//Método de instancia
public void imprime(){
  System.out.println("Nombre = " + nombre + "RFC = " + RFC);
  }
} // Fin de la clase empleado

public class UsoEmpleado{
 public static void main(String[ ] args){
    Empleado e1 = new Empleado("Miguel Cervantes", "1600");
    Empleado e2 = new Empleado("Miguel Quevedo", "1650");
    //Invocación al método de clase de la clase Empleado
    Empleado.reportarEmpleados();
        }
}
```

Referencias

- Booch Grady. Análisis y diseño orientado a objetos, con aplicaciones. 2ª Ed., Pearson Educación. Santa Clara, California, 1996.
- O'Docherty Mike. ObjectOriented Analysis and Design. Understanding System Development with UML 2.0.John Wiley &Sons. England.

Programación Orientada a Objetos
Clases y Objetos

Rafael Aguilar García

La clase Cuenta Bancaria

- Diseñaremos una clase para manejar una cuenta bancaria.
  - La cuenta tiene un saldo
  - Podemos hacer operaciones con el saldo:
    - Hacer depósitos
    - Hacer retiros
    - Desplegar la información del saldo actual

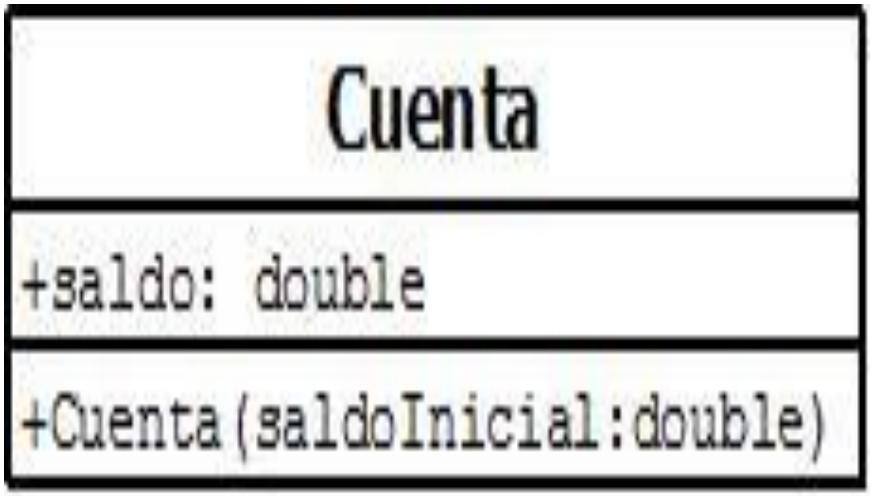
La clase Cuenta Bancaria

- Atributos:
  - Saldo en pesos
- Operaciones:
  - **Depositar** una cantidad (requiere la cantidad)
  - **Retirar** una cantidad (devuelve la cantidad)
  - **Consultar** saldo actual (devuelve el saldo
- Condiciones:
  - El saldo debe ser mayor o igual que 0 pesos

Clase Cuenta Bancaria

1ª Versión

Diagrama UML de la clase Cuenta



Nombre de la clase

Atributos

Constructor

+ = acceso público: cualquier otra clase tiene acceso

Constructor(UML)

+Cuenta(saldoInicial: double)

Tipo de dato del argumento

Nombre del argumento

Constructor (mismo nombre que la clase)

Código Java de la clase Cuenta

```
public class Cuenta {
        // Atributos
        public double saldo;

         // Constructor: inicializa objetos de la clase Cuenta
         public Cuenta(double saldoInicial){
                saldo = saldoInicial;
         }
 }
```

Archivo: Cuenta.java

Uso de la clase Cuenta

```
public class UsarCuenta {
        public static void main(String [ ] args){
                //Creamos el objeto cta1 con el constructor de la clase,
                // con un saldo inicial de 100.0 pesos
                Cuenta cta1 = new Cuenta (100.0);
                //Creamos el objeto cta2 con el constructor de la clase,
                // con un saldo inicial de 235.60 pesos
                Cuenta cta2 = new Cuenta (235.60);
                 //Depositamos 30.5 pesos a cta1
                 cta1.saldo = cta1.saldo + 30.5;
                 //Depositamos 65.83 pesos a cta2
                 cta2.saldo = cta2.saldo + 65.83;
                 //Imprimimos los saldos de ambas cuentas
                 System.out.println("Saldo cta1 = " + cta1.saldo);
                 System.out.println("Saldo cta2 = " + cta2.saldo);
```

Uso de la clase Cuenta (cont.)

```
            //Retiramos 500 pesos de cta1
            cta1.saldo = cta1.saldo - 500;
            //Retiramos 500 pesos de cta2
            cta2.saldo = cta2.saldo - 500;
            //Imprimimos los saldos de ambas cuentas
            System.out.println("Saldo cta1 = " + cta1.saldo);
            System.out.println("Saldo cta2 = " + cta2.saldo);
    } // Fin de método main
} // Fin de la clase
```

Archivo: UsarCuenta.java

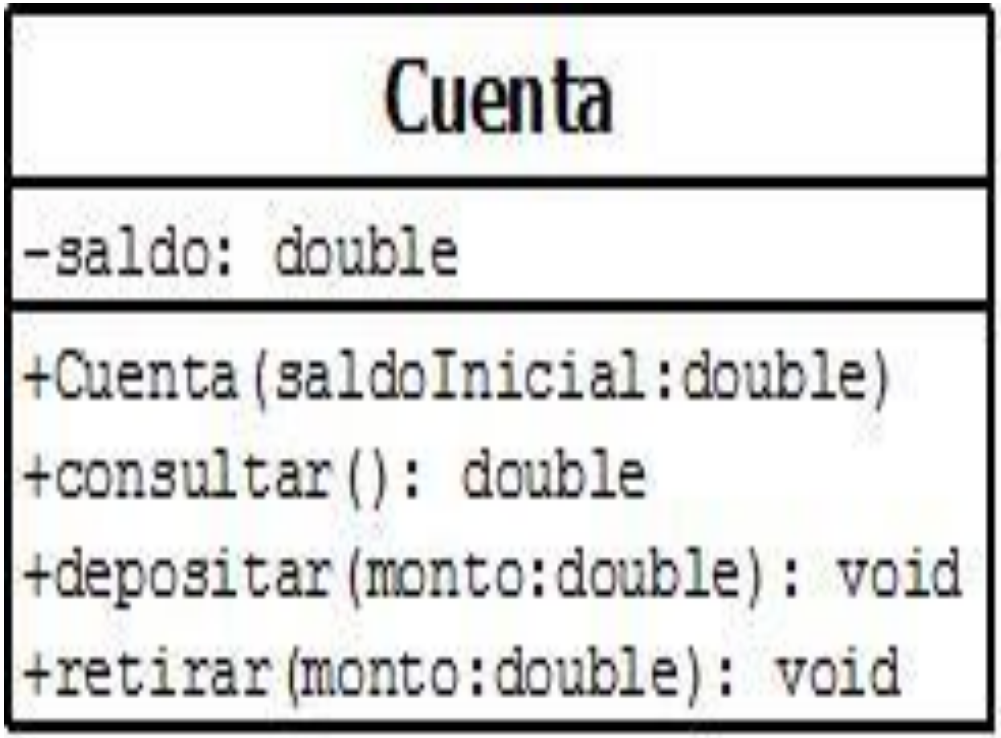
Errores en la clase Cuenta

- El saldo final de las dos cuentas es negativo; esto no cumple con la condición establecida al inicio de que el saldo debe ser mayor o igual a cero pesos.
- El saldo no debe tener un acceso público; debe estar oculto a los usuarios de la clase.
- Las operaciones deben encapsularse, no deben ser realizadas por los usuarios de la clase.

Clase Cuenta Bancaria

2ª Versión

Diagrama UML para la clase Cuenta Bancaria



Nombre de la clase

Atributos

Constructor

Operaciones o métodos

- = acceso privado: solo la propia clase tiene acceso

+ = acceso público: cualquier otra clase tiene acceso

Operaciones (UML)

+Cuenta(saldoInicial: double)

Tipo de dato del argumento

Nombre del argumento

Constructor (mismo nombre que la clase)

Operaciones (UML)

+consultar( ): double

Nombre del método

Sin argumentos

Tipo de dato que retorna el método

Operaciones (UML)

+depositar(monto: double): void

Nombre del método

Nombre del argumento

Tipo de dato del argumento

No hay tipo de dato de retorno

Operaciones (UML)

+retirar(monto: double): void

Nombre del método

Nombre del argumento

Tipo de dato del argumento

No hay tipo de dato de retorno

Nota: El método *retirar* tiene que verificar que el monto a retirar no sea mayor que el saldo

Código Java – clase Cuenta

```
public class Cuenta {
        // Atributo con acceso privado
        private double saldo;

         // Constructor: inicializa objetos de la clase Cuenta
         public Cuenta(double saldoInicial){
                saldo = saldoInicial;
         }

         //Métodos
         // Consultar: devuelve el saldo actual
         double consultar( ){
              return saldo;
         }
```

Código Java de la clase Cuenta

```
        //Métodos
        // Depositar: incrementa el saldo en el monto indicado
        void depositar(double monto ){
            saldo = saldo + monto;
        }
        // Retirar: decrementa el saldo en el monto indicado siempre
        // que el monto sea mayor o igual al saldo
        void retirar(double monto ){
            if ( saldo >= monto ) {
                saldo = saldo - monto;
            }
            else {
                System.out.println("Saldo insuficiente");
            }
        }
} // Fin de la clase Cuenta
```

Uso de la clase Cuenta

```
public class UsarCuenta {
        public static void main(String [ ] args){
                //Creamos el objeto cta1 con el constructor de la clase,
                // con un saldo inicial de 100.0 pesos
                Cuenta cta1 = new Cuenta (100.0);
                //Creamos el objeto cta2 con el constructor de la clase,
                // con un saldo inicial de 235.60 pesos
                Cuenta cta2 = new Cuenta (235.60);
                 //Depositamos 30.5 pesos a cta1
                 cta1.depositar(30.5);
                 //Depositamos 65.83 pesos a cta2
                 cta2.depositar(65.83);
                 //Imprimimos los saldos de ambas cuentas
                 System.out.println("Saldo cta1 = " + cta1.consultar( ));
                 System.out.println("Saldo cta2 = " + cta2.consultar( ));
```

Uso de la clase Cuenta (cont.)

```
            //Retiramos 500 pesos de cta1
            cta1.retirar(500);
            //Retiramos 500 pesos de cta2
            cta2.retirar(500);
            //Imprimimos los saldos de ambas cuentas
            System.out.println("Saldo cta1 = " + cta1.consultar( ));
            System.out.println("Saldo cta2 = " + cta2.consultar( ));
    } // Fin de método main
} // Fin de la clase
```

Archivo: UsarCuenta.java

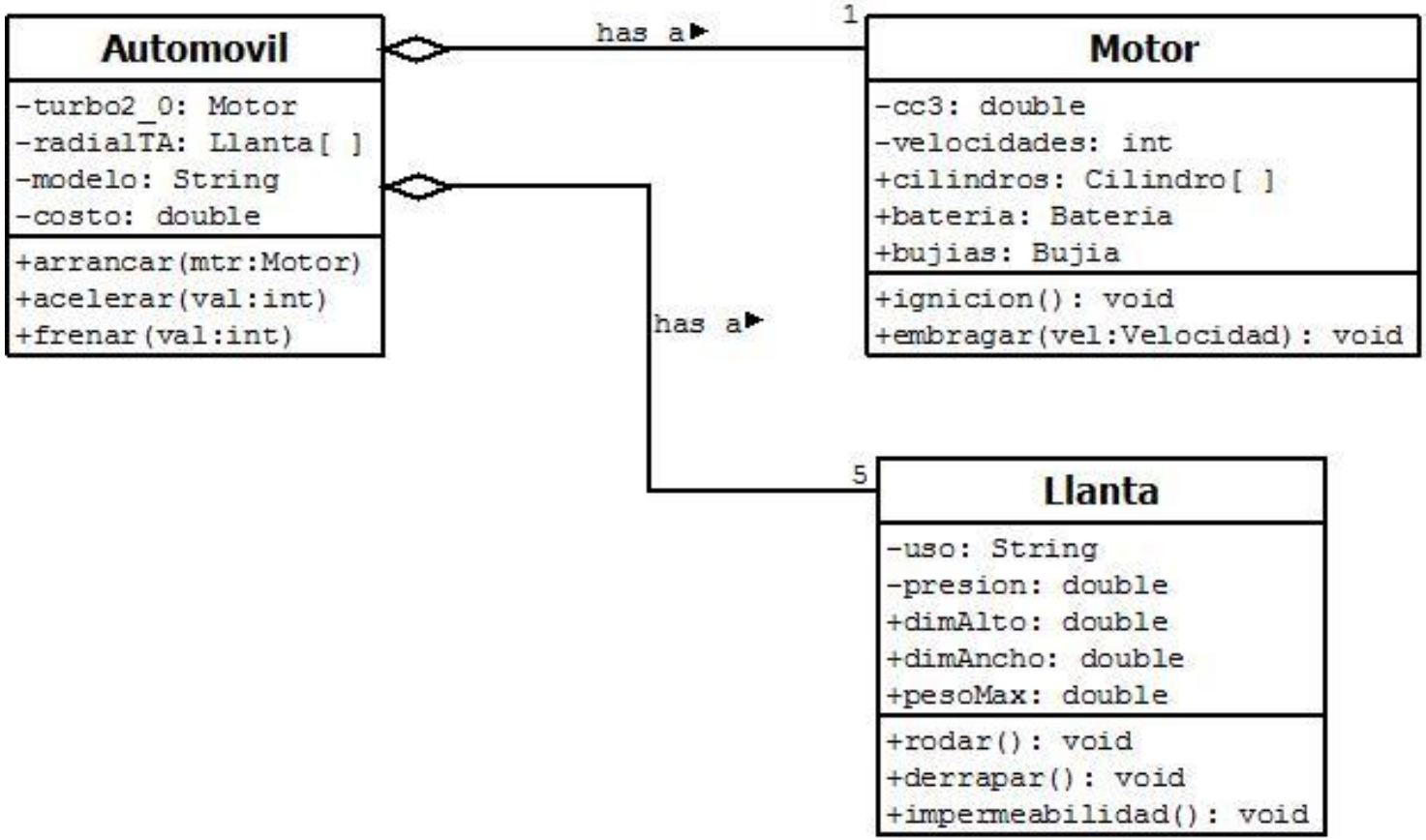
Programación Orientada a Objetos
Agregación de clases

M. en C. Rafael Aguilar García

Agregación de Clases

- Consiste en definir una clase usando objetos de otra u otras clases.
- Por ejemplo:

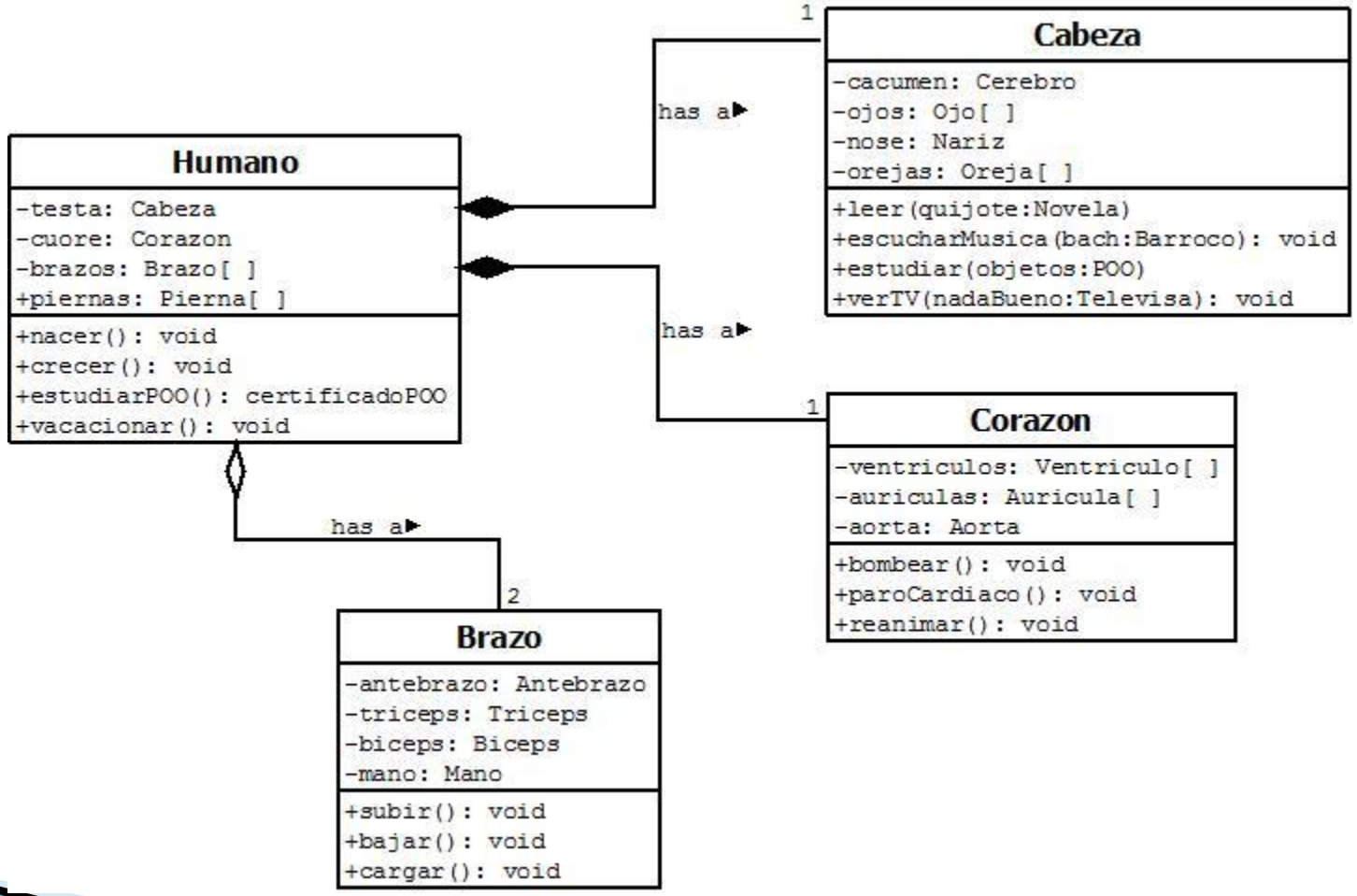
Agregación de Clases



Composición de Clases

- Consiste en definir una clase principal usando objetos de otra u otras clases. Las clases usadas no pueden ser separadas de la clase principal.
- Por ejemplo:

Composición de Clases

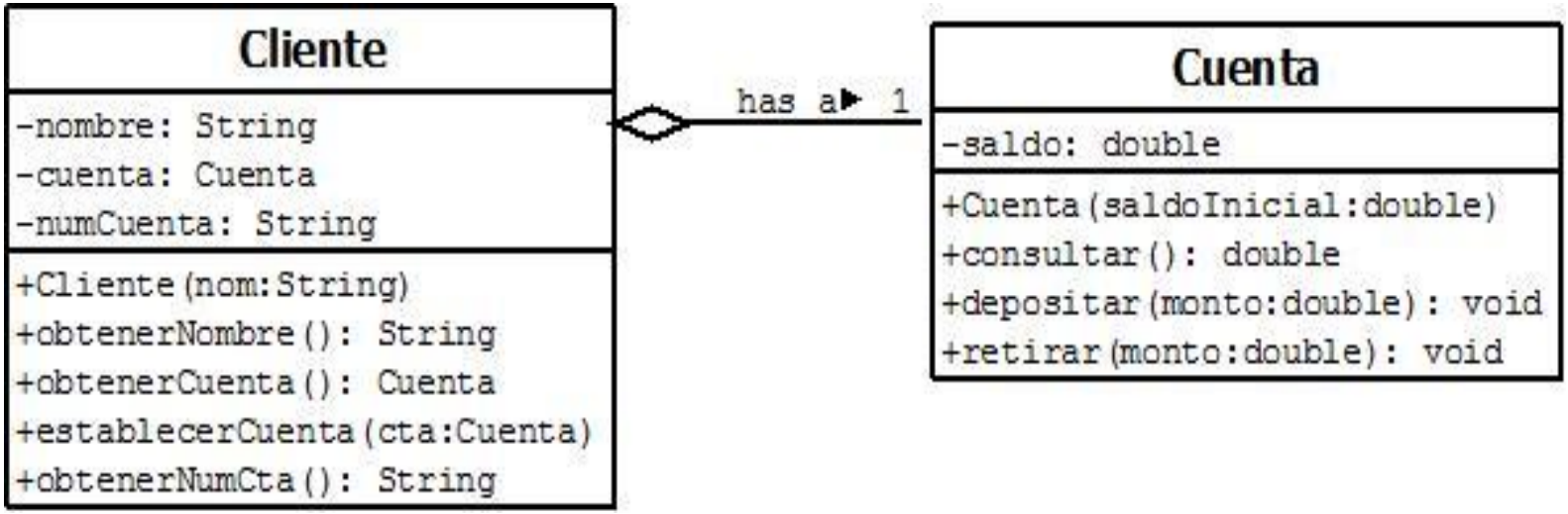


Agregación de Clases

- Ejemplo:
  - Usaremos la clase Cuenta
  - Crearemos la clase Cliente
  - Relacionaremos ambas clases: Un Cliente tiene una Cuenta

Agregación de Clases

- Agregación: Cliente – Cuenta



Código Java – clase Cuenta

```
public class Cliente {
        // Atributo con acceso privado
        private String nombre;
        private Cuenta cuenta;
        private String numCta;

         // Constructor: inicializa objetos de la clase Cliente
         public Cliente(String nombre, String numCta){
                this.nombre = nombre;
                this.numCta = numCta;
         }
         //Métodos
         // obtenerNombre: devuelve el  nombre
         public String  obtenerNombre( ){
              return nombre;
         }
```

Clase Cliente (2)

```
        //Métodos
        // obtenerCuenta: devuelve el objeto de la clase Cuenta
        public Cuenta obtenerCuenta( ){
            return cuenta;
        }

        // establecerCuenta: establece el objeto de la clase Cuenta
        public
establecerCuenta(Cuenta cta ){
            cuenta = cta;
        }
        // obtenerNombre: devuelve el  numero de cuenta
        public String  obtenerNumCta( ){
            return numCta;
        }

} // Fin de la clase Cuenta
```

Uso de la clases Cuenta y Cliente

```
public class Bankito{
        public static void main(String [ ] args){
            Cuenta ctaAux;
            //Creamos el cliente "El Quijote"
            // con un num. de cuenta: QX400
            Cliente qx = new Cliente ("El Quijote", "QX400");
            //Creamos  su cuenta con saldo de 550.45
            qx.establecerCuenta(new Cuenta (550.45));
            //Obtenemos la cuenta del Quijote
            //y la asignamos a la referencia ctaAux
            ctaAux = qx.obtenerCuenta( );
             //Ahora depositamos 300.55 pesos a cuenta del Quijote
             ctaAux.depositar(300.55);
             //Ahora imprimimos toda la información del Quijote
             System.out.println("Nombre Cliente:" + qx.obtenerNombre( ));
             System.out.println("Num cuenta:"+ qx.obtenerNumCta( ));
```

Uso de la clase Cuenta (cont.)

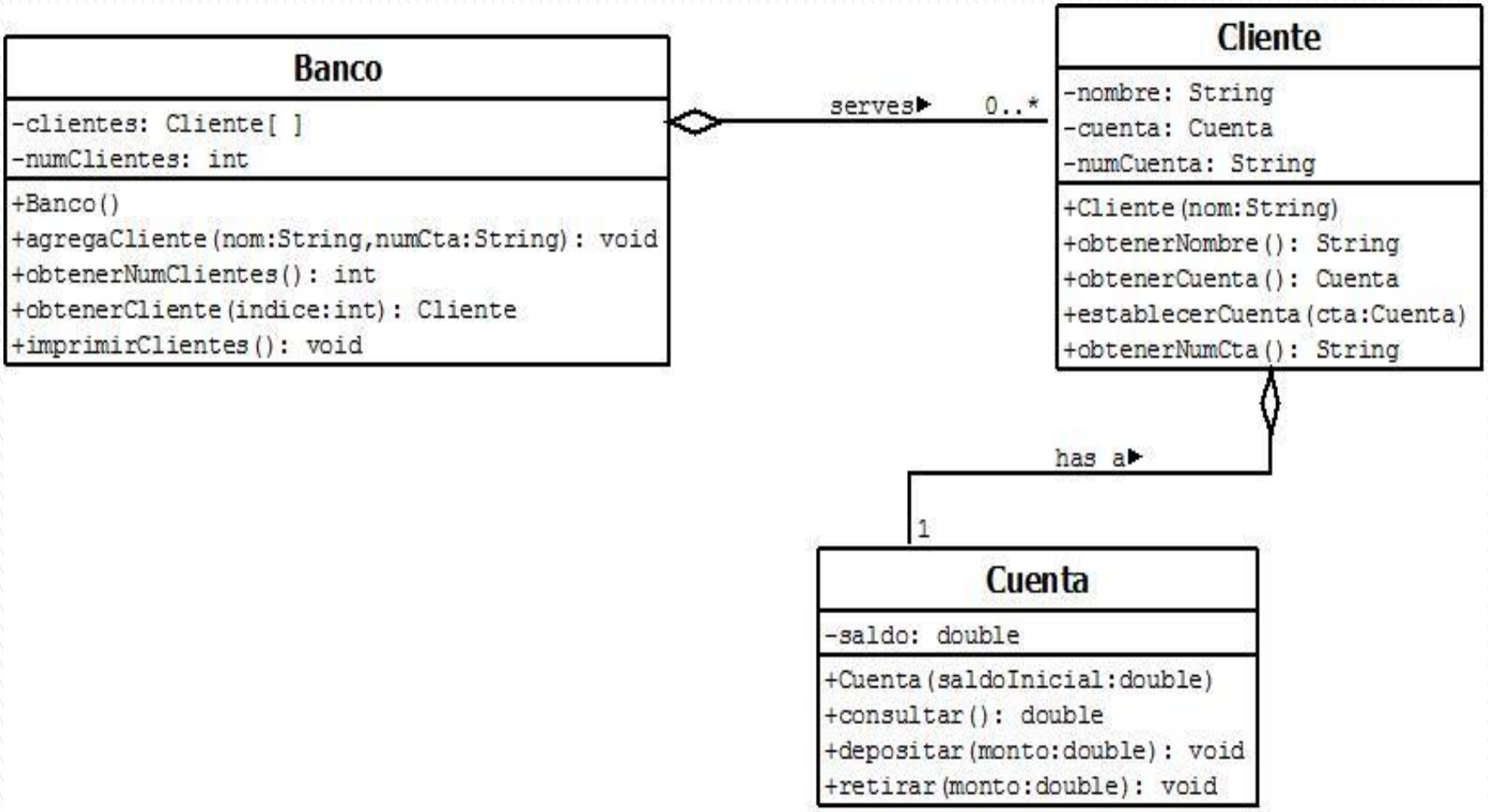
```
        System.out.println("Saldo:"+ ctaAux.consultar( ));

    } // Fin de método main
} // Fin de la clase
```

Archivo: Bankito.java

Ejercicio 1

- Agrega lo siguiente al programa anterior:
  - Crea el cliente “Sancho Panza” y asígnale un numero de cuenta “SPANZA”
  - Asígnale una cuenta con un saldo inicial de 100.35 pesos.
  - Realiza un retiro de 100 pesos.
  - Reporta toda la información de Sancho en pantalla.
  - Nota: La información del Quijote debe permanecer inalterada.

Ejercicio 2: Escribe el código Java para el siguiente diagrama UML.



Ejercicio 3

- Modifiqua el código correspondiente al diagrama anterior para realizar lo siguiente:
  - Asigna un saldo inicial a la cuenta de cada uno de los clientes e imprima toda la información de los clientes.
  - Realiza operaciones de movimientos en los saldos de las cuentas y a continuación imprima toda la información de los clientes.

Programación Orientada a Objetos
Conceptos de Herencia
Herencia y Agregación

M. en C. Rafael Aguilar García

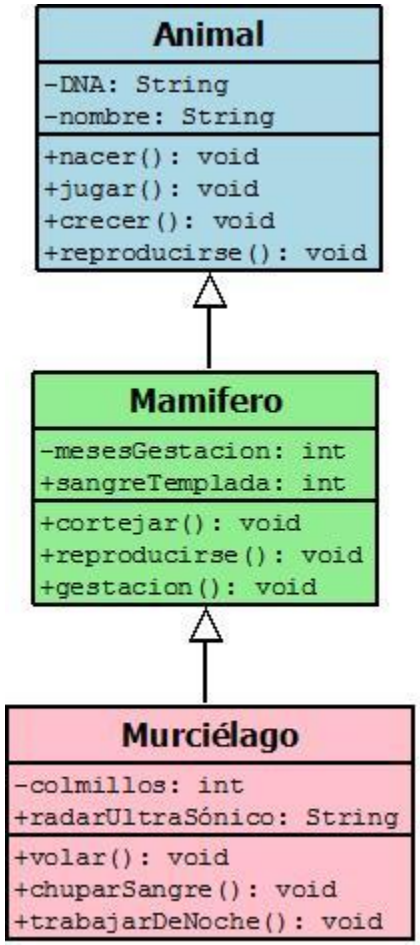
Herencia de Clases

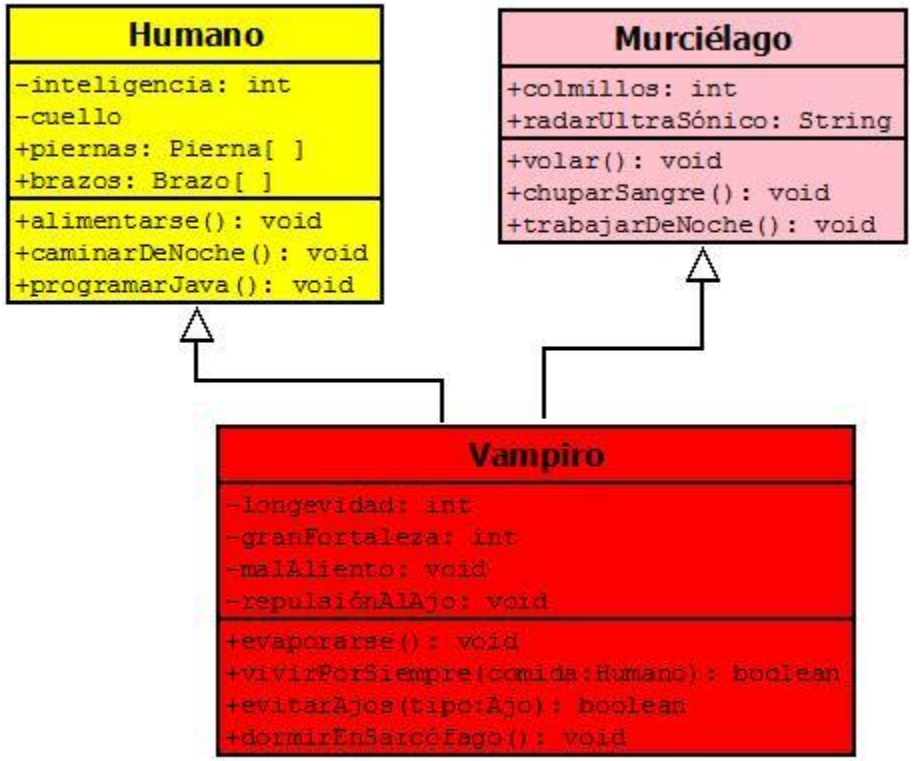
- La herencia nos permite derivar una nueva clase a partir de una clase existente.
- La clase existente es conocida como superclase, clase base, o clase madre.
- La clase derivada también es conocida como clase hija o subclase.

Herencia de Clases

- En la herencia partimos de un modelo general o global el cual se irá transformando en modelos más especializados.
- La herencia junto con la agregación son los mecanismos de reutilización de código de la POO.

Herencia de Clases – Tipos de Herencia



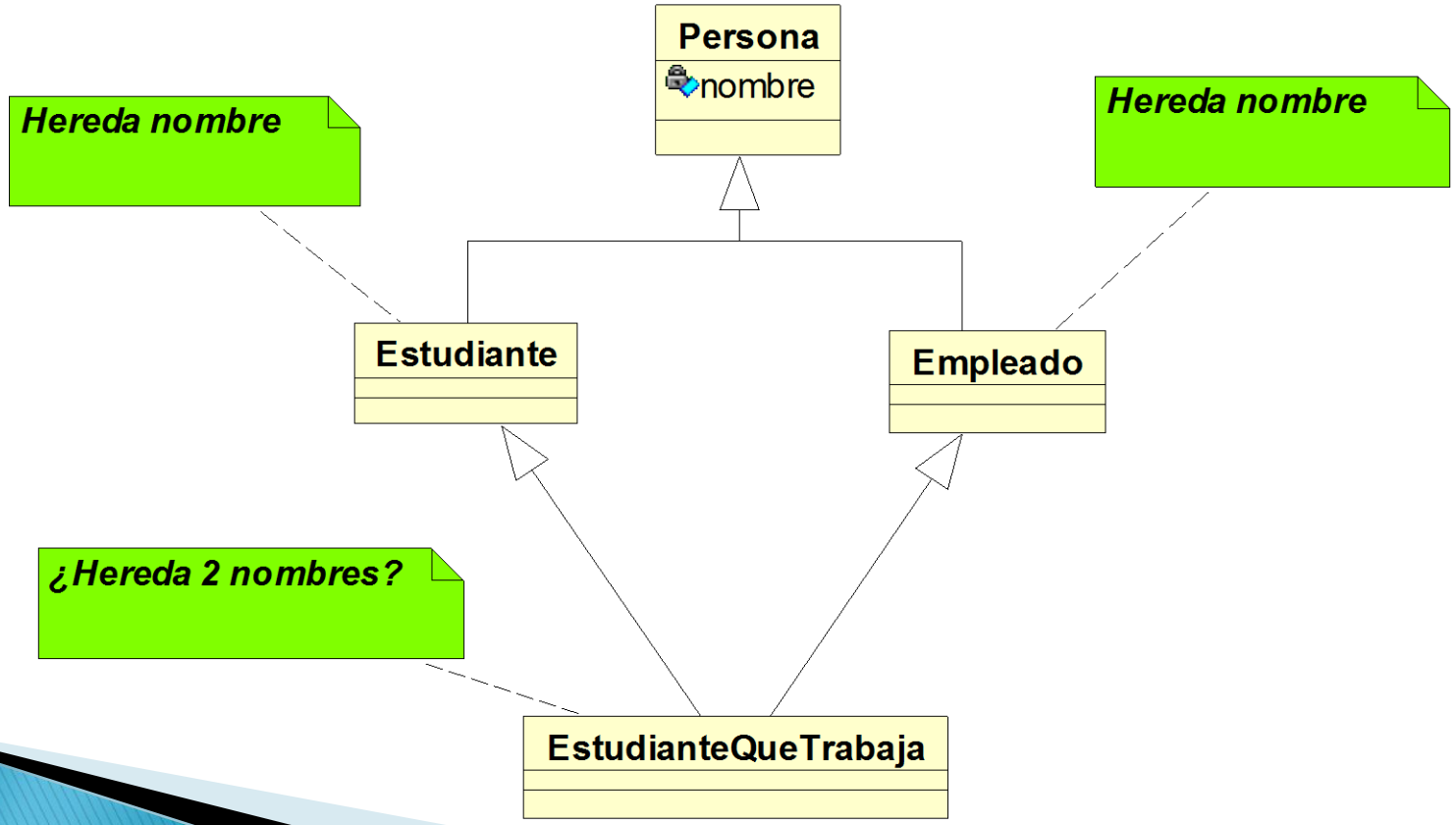


Herencia Simple

Herencia Múltiple

Herencia de Clases

- En Java, la herencia múltiple no está permitida, esto con el propósito de evitar ambigüedades.

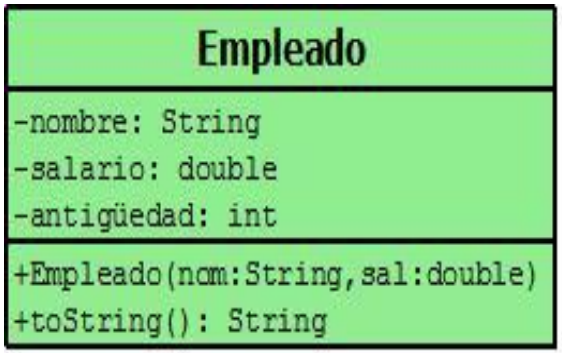


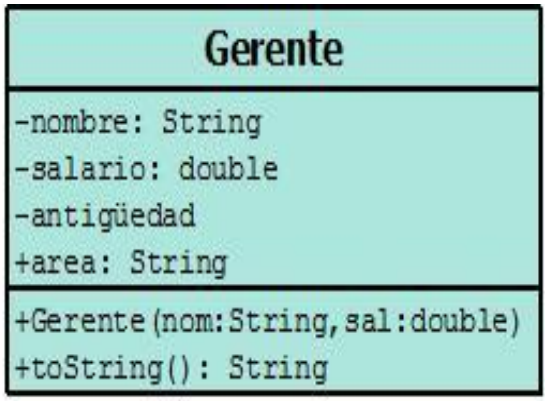
Herencia de Clases

- Por ejemplo:
  - Queremos un modelo para la clase Empleado
  - Queremos también un modelo para la clase Gerente
  - Podemos decir que un Gerente es un Empleado con características adicionales o especiales.

Herencia de Clases

- Podemos representar ambos modelos de la siguiente forma:

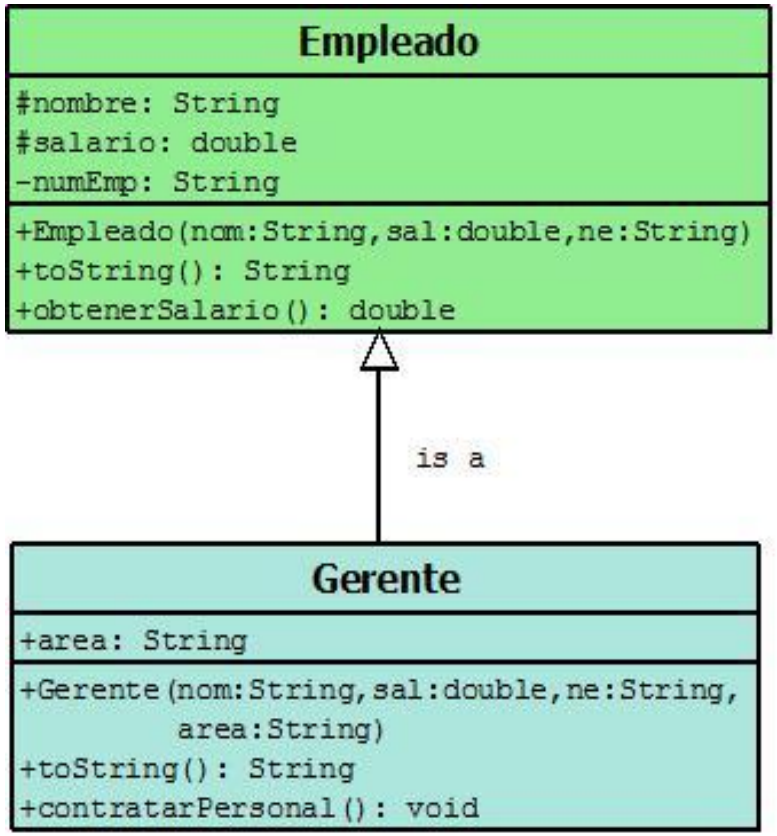




Herencia de Clases

- El esquema anterior repite código, esto es, la clase Gerente tiene los atributos de la clase Empleado.
- Para programar con eficiencia debemos reutilizar código, en este caso mediante la herencia.
- En este ejemplo tenemos la relación: Un Gerente **es un** Empleado.

Herencia de Clases

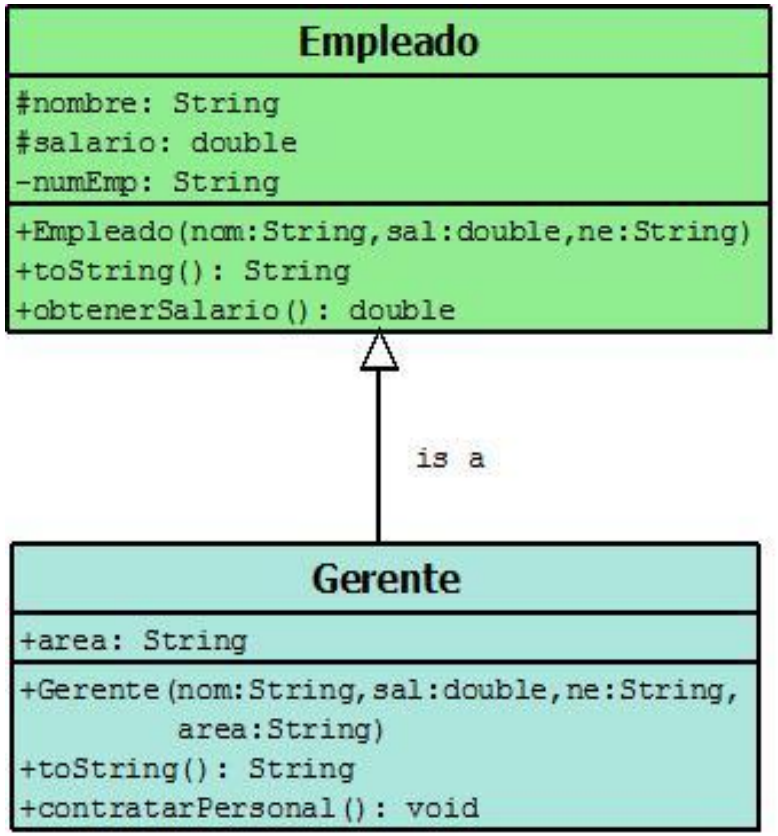


Sintaxis:

```
public class Empleado{
    . . .
}
```

```
public class Gerente extends Empleado{
. . .
}
```

Herencia de Clases



Clase Base

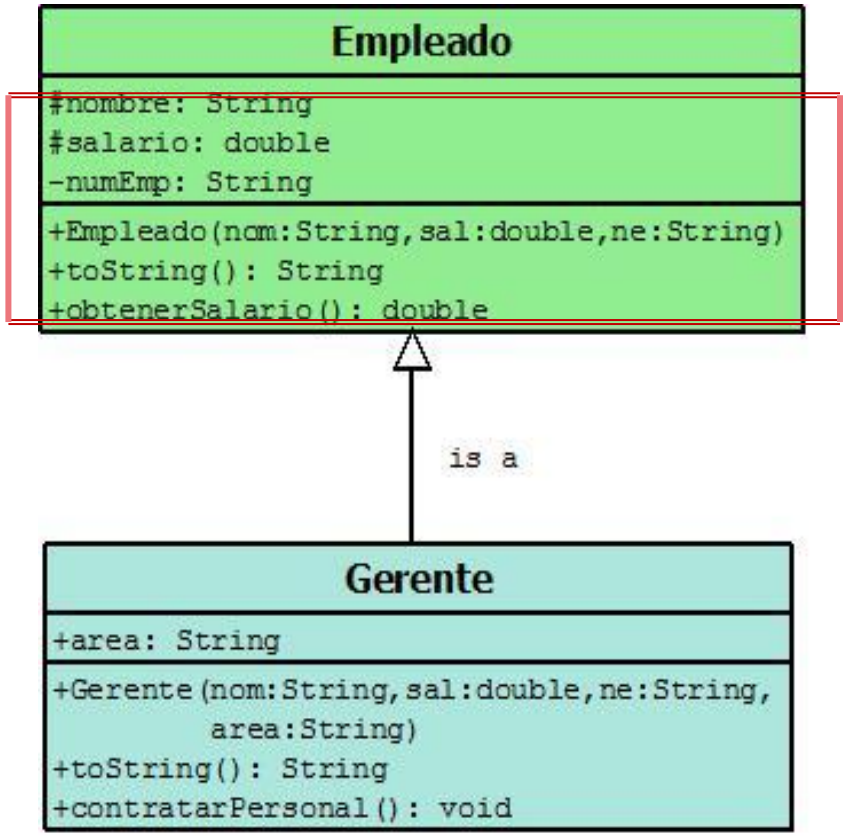
Gerente **es un** Empleado

Clase Derivada

= acceso protegido: solo la misma clase base y las clases derivadas tienen acceso

- = acceso privado: solo la clase base tiene acceso. Las clases derivadas no tienen acceso.

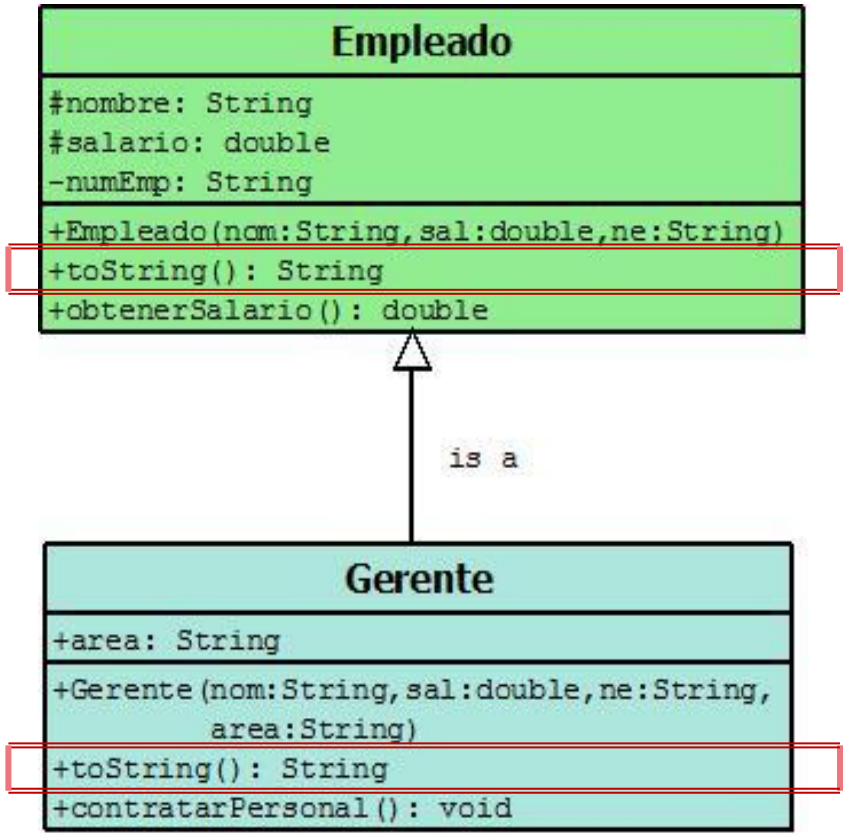
Herencia de Clases



Herencia directa:

La subclase hereda los atributos y métodos de su clase base, pero no puede acceder a los declarados como *private.*
Puede acceder a *protected* , *public*, y sin modificador (default).

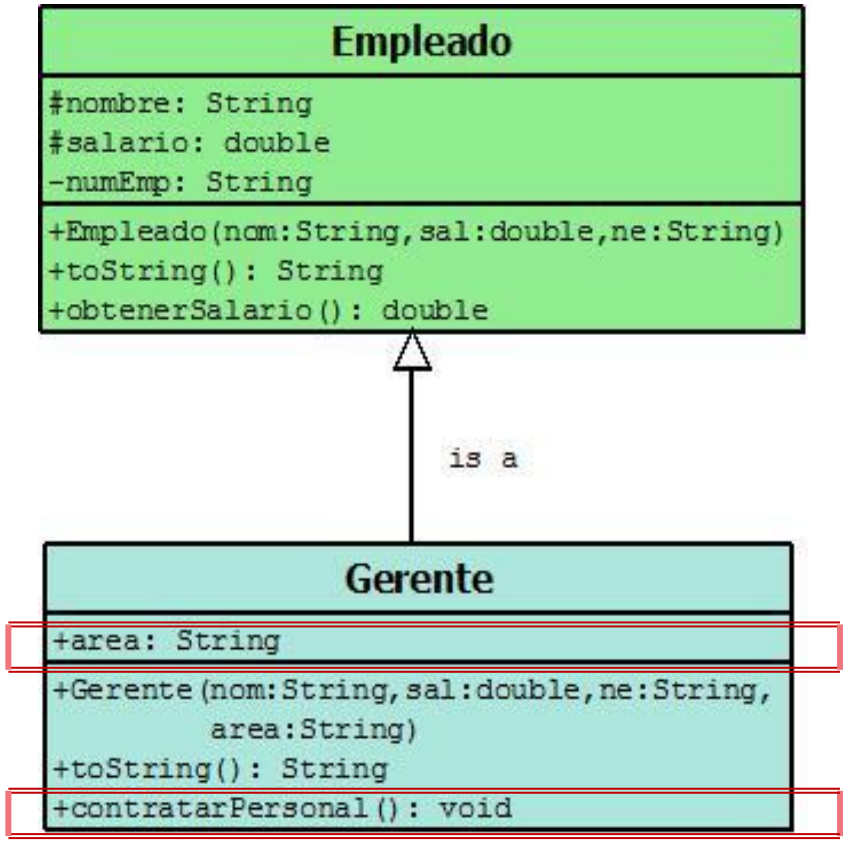
Herencia de Clases



Redefinición de Métodos: (*Overriden*)

La subclase puede redefinir métodos de su clase base (v.g. toString( )), esto es, da su propia versión (comportamiento).

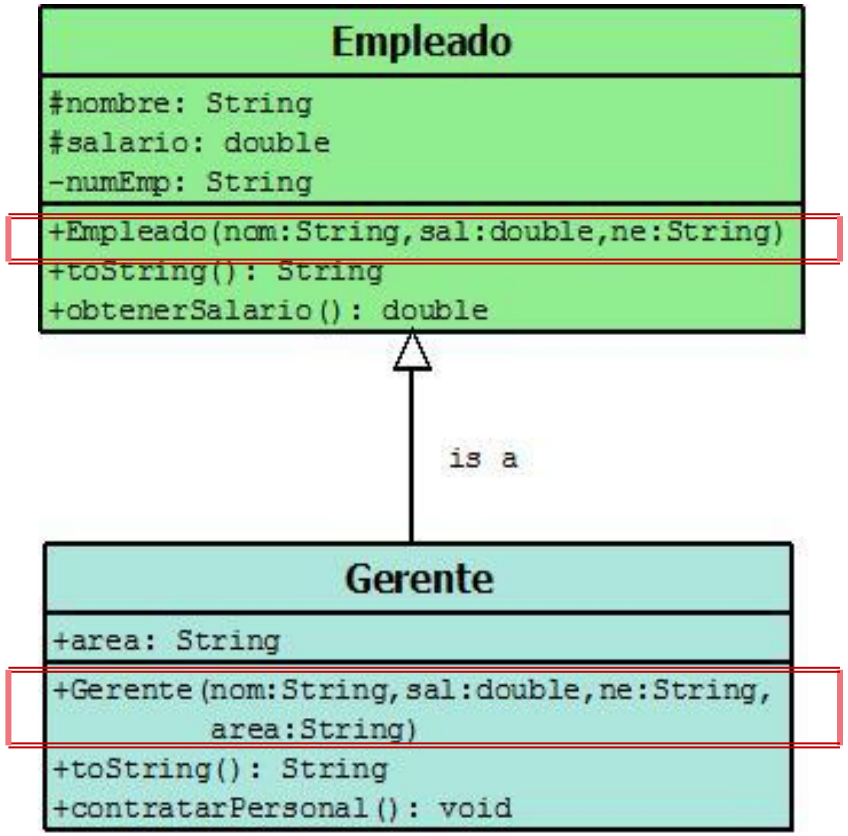
Herencia de Clases



Especialización:

La subclase puede agregar nuevos atributos y/o métodos.

Herencia de Clases



Constructores:

En una jerarquía de clases:

- Los constructores **no** son heredados de la superclase a la subclase.
- Los constructores deben ser invocados a partir de la clase raíz (la clase más alta de la jerarquía) continuando en orden descendiente por las subclases.
- Por lo tanto, el constructor de la Clase **Gerente** debe invocar (ya se (explícita o implícitamente) al constructor de la clase **Empleado**

Herencia de Clases

- La subclase puede acceder a atributos o métodos de su clase base usando la palabra reservada **super** (siempre y cuando tenga el acceso permitido):

  Sintáxis: **super.***dentificador_atributo;*
  super.*dentificador_método(arg1,arg2,…);*

- La subclase puede invocar al constructor de su clase base de forma explícita usando:

  super(arg1, arg2,…);

  El constructor invocado será aquel que coincida en el número y tipo de los argumentos en la superclase.

Herencia de Clases

- Si la subclase no invoca de forma explícita al constructor de su clase base, entonces la clase base deberá incluir un constructor sin argumentos (*default*), de otra forma el compilador indicará un error.
-

Código Java: Empleado.java

```
public class Empleado{
        protected String nombre;
        protected double salario;
        private String numEmpleado;

        public Empleado(String nom, double sal, String ne){
                nombre = nom;
                salario = sal;
                numEmpleado = ne;
        }

        // El método toString es un método de la clase Object, la
        // clase empleado está redefiniendo este método:
        public String toString(){
        return "Nombre: " + nombre +  " Salario: " + salario + " Numero de
Empleado: " + numEmpleado;
        }
}
```

Código Java: Gerente.java

```
public class Gerente extends Empleado{
   private String area;

   public Gerente(String nom, double sal, String ne, String a){
         //Invocamos al constructor de la superclase con sus 3 argumentos requeridos
         super(nom, sal, ne);
         //Continuamos con la construcción del objeto de la clase Gerente
         area = a;
    }
   // El método toString es un método de la clase Object, la clase Gerente está
   // redefiniendo este método.
   // Usamos reutilización del método toString definido en la superclase:
   public String toString(){
         return super.toString( ) + " Gerente del area de: " + area;
   }

   public void contratarPersonal(){
      System.out.println("Las contrataciones en el area de: " + area + " fueron 2" );
    }
```

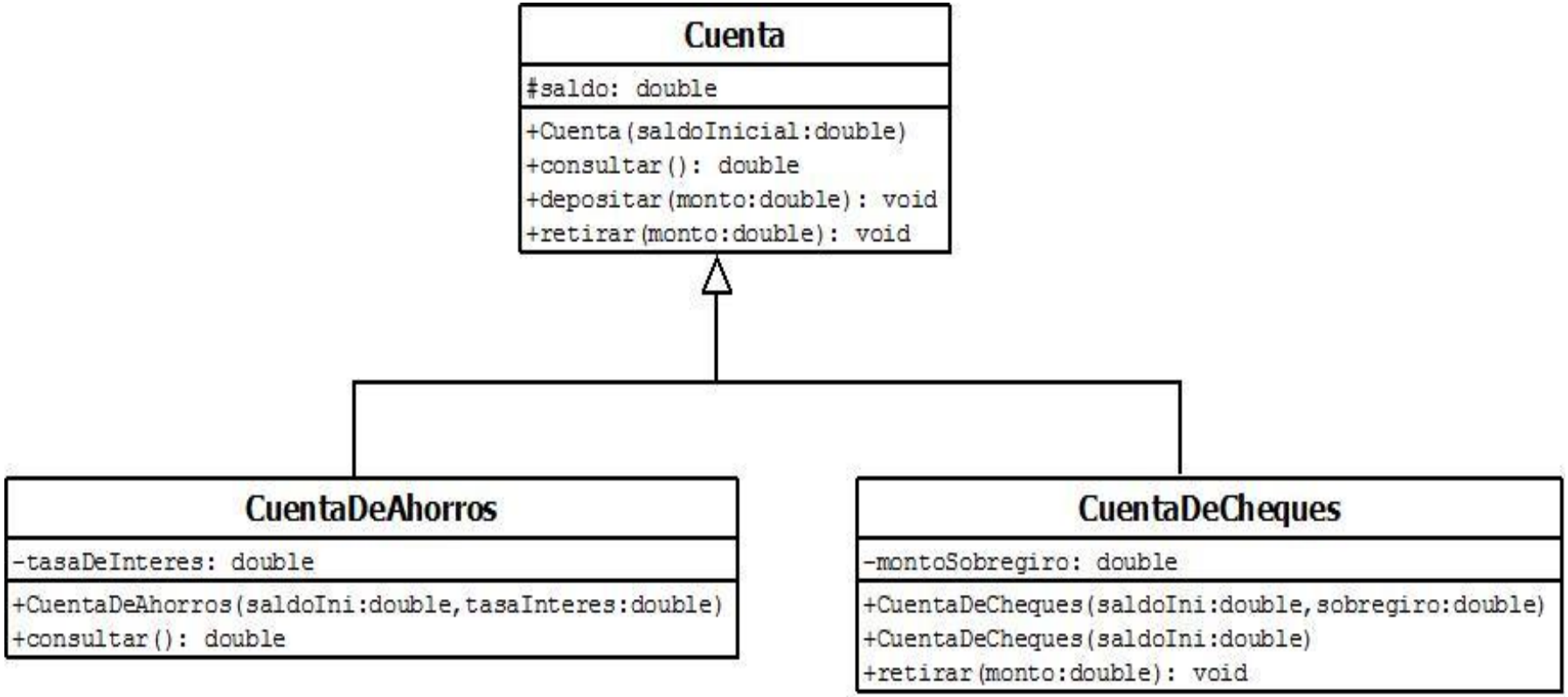
Código Java: Gerente.java (cont.)

```
    // El método cambiarSalario ilustra el acceso de la clase Gerente a los atributos
    // protegidos de su clase base
     public void cambiarSalario(double nuevo){
            salario = nuevo;
     }
} //Fin de la clase
```

Código Java: UsoHerencia.java

```
public class UsoHerencia{
  public static void main(String[ ] args){
    Empleado e1 = new Empleado("Barak Obama", 3650.45, "00001");
    Gerente g1 = new Gerente("Silvio Berlusconi", 2642.32, "0002", "Espectaculos");

    //El método toString() de la clase Empleado se invoca de forma implícita:
     System.out.println(e1);

    //El método toString() de la clase Gerente se invoca de forma implícita:
     System.out.println(g1);
     g1.contratarPersonal();

     g1.cambiarSalario(3200.46);
     System.out.println(g1);

     //Segundo constructor para la clase Gerente:
     //  Gerente g2 = new Gerente("Ventas");
     }
}
```

Ejercicio 1. Escribe el código Java para el siguiente diagrama UML



Ejercicio 1. Lineamientos

- Crea las subclases CuentaDeAhorros y CuentaDeCheques
- Una CuentaDeAhorros gana intereses de forma mensual, según la formula:

  saldo = **saldo** + (**tasaDeInteres** * **saldo**)
- Redefine el método consultar en la clase cuentaDeAhorros de forma que actualice el saldo conforme a la formula anterior.

Ejercicio 1. Lineamientos (cont.)

- Una **CuentaDeCheques** permite que un cliente haga cualquier número de depósitos y retiros. Como medida de protección, el banco permite un monto fijo por sobregiro, de forma que el cliente puede tener saldos negativos pero no por debajo del monto de sobregiro de protección (**montoSobregiro**).

- Diseña el código de forma que soporte el siguiente programa:

```
public class HerenciaCuentas{
  public static void main(String[ ] args){
    //Creamos una instancia de CuentaDeAhorros
    CuentaDeAhorros ca1 = new CuentaDeAhorros(1000, 0.3);
   //Creamos una instancia de CuentaDeCheques
   CuentaDeCheques cch1 = new CuentaDeCheques(1000, 1000);
   //Consultamos la cuenta de ahorros
   System.out.println("cuenta de ahorros: " + ca1.consultar());
  //Consultamos la cuenta de cheques
   System.out.println("cuenta de cheques, saldo: " + cch1.consultar());
   cch1.retirar(1000);
   System.out.println("cuenta de cheques, saldo: " + cch1.consultar());
   cch1.retirar(500);
   System.out.println("cuenta de cheques, saldo: " + cch1.consultar());
   cch1.consultarSobregiro();
   cch1.retirar(499);
   cch1.consultarSobregiro();
   cch1.retirar(2);
   }
}
```

La salida del código anterior es:

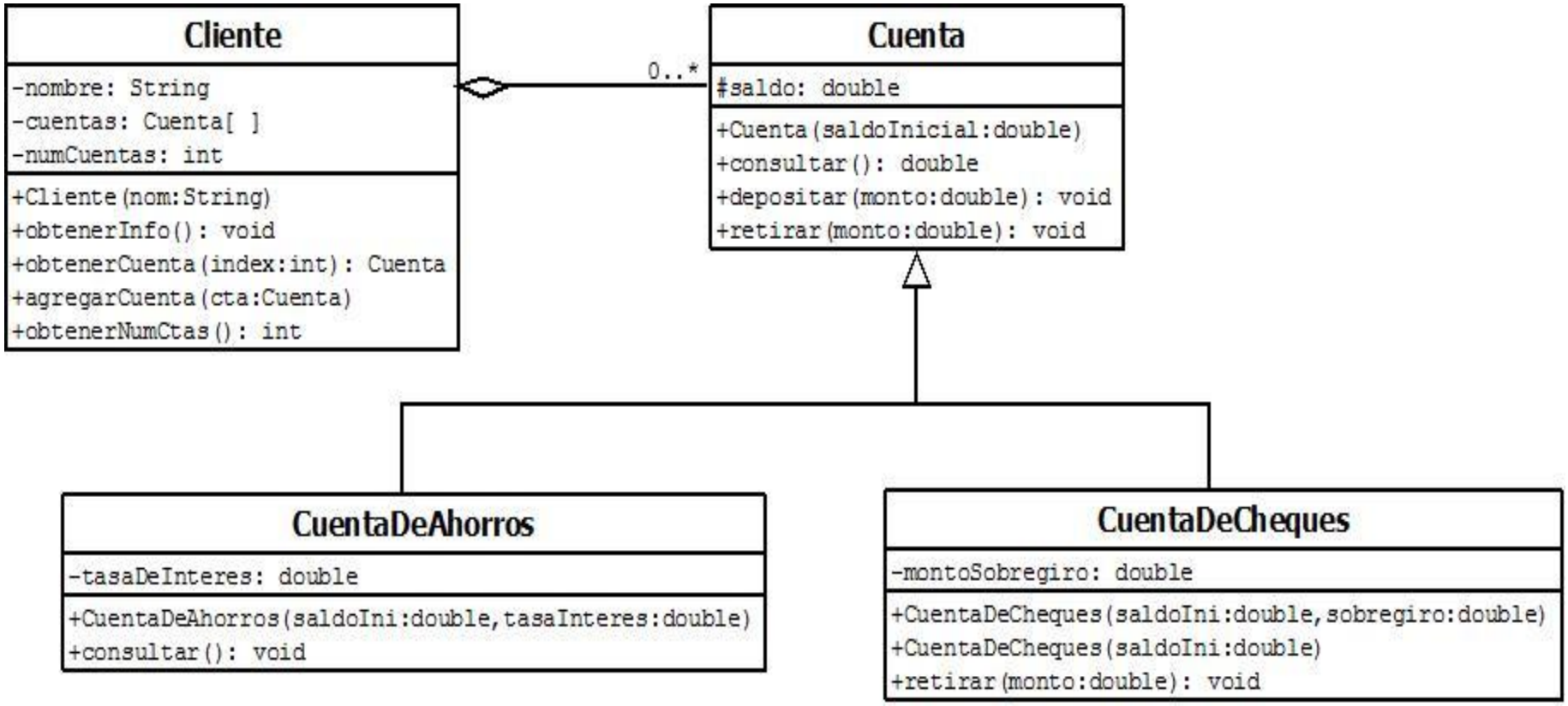
Símbolo del sistema

```
C:\ESCOM\CEC\Codigo\Herencia\Cuentas>java HerenciaCuentas
cuenta de ahorros: 1300.0
cuenta de cheques, saldo: 1000.0
cuenta de cheques, saldo: 0.0
cuenta de cheques, saldo: 0.0
Su sobregiro es: 500.0
Su sobregiro es: 1.0
Lo sentimos, su transaccion fue cancelada.
El monto requerido: [2.0] supera su saldo mas su sobregiro establecido: [1.0]

C:\ESCOM\CEC\Codigo\Herencia\Cuentas>
```

Ejercicio 2. Integrando la clase Cliente a las subclases de Cuenta

- Escribe el código del siguiente diagrama de clases UML

UML: Agregación: Cliente – Cuenta; Herencia: Subclases de Cuenta



Lineamientos del ejercicio 2

- La clase **Cliente** puede ahora contener varios tipos de cuenta, que almacenará en un arreglo de objetos (tipo clase **Cuenta)** denominado **cuentas**.
- Un **Cliente** puede agregar una **CuentaDeAhorros** o una **CuentaDeCheques** a su arreglo **cuentas**, mediante el método **agregarCuenta** que toma como argumento un objeto (subclase) de la clase **Cuenta**. Esto incrementará el valor de **numCuentas**

Lineamientos del ejercicio 2

- Para recuperar una cuenta del **Cliente** usamos el método **obtenerCuenta** que requiere un número entero y devolverá la **Cuenta** correspondiente del arreglo de cuentas.
- El método **obtenerInfo** reporta el nombre del cliente y todas las cuentas que tenga asociadas, desplegando el tipo de cuenta (ahorros y/o cheques) así como los saldos respectivos.

Lineamientos del ejercicio 2

- El diseño de las clases debe soportar el siguiente código:

```
public class UsoTodo{
        public static void main(String[] args){
                //Creamos cuentas de ahorro
                CuentaDeAhorros cda1 = new CuentaDeAhorros(1000, 0.1);
                CuentaDeAhorros cda2 = new CuentaDeAhorros(2000, 0.2);
                CuentaDeAhorros cda3 = new CuentaDeAhorros(3000, 0.3);
                //Creamos cuentas de cheques
                CuentaDeCheques cdc1 = new CuentaDeCheques(1000, 100);
                CuentaDeCheques cdc2 = new CuentaDeCheques(2000, 200);
                CuentaDeCheques cdc3 = new CuentaDeCheques(3000, 300);
                //Creamos 3 clientes
                Cliente cli1 = new Cliente("Quijote");
                Cliente cli2 = new Cliente("Sancho");
                Cliente cli3 = new Cliente("Rocinante");
                //Agregamos cuentas al cliente Quijote
                cli1.agregarCuenta(cda1);
                cli1.agregarCuenta(cdc1);
                //Agregamos cuentas al cliente Sancho
                cli2.agregarCuenta(cda2);
                cli2.agregarCuenta(cdc2);
                //Agregamos cuentas al cliente Rocinante
                cli3.agregarCuenta(cda3);
                cli3.agregarCuenta(cdc3);
                //Obtenemos la informacion de cada cliente
                cli1.obtenerInfo();
                cli2.obtenerInfo();
                cli3.obtenerInfo();
        }
}
```

Salida en pantalla:

```
C:\ESCOM\CEC\Codigo\Herencia\Agregacion_Herencia>java UsoTodo


Nombre del cliente: Quijote
Informacion de sus cuentas
Cuenta de Ahorros        saldo: 1100.0
Cuenta de Cheques        saldo: 1000.0


Nombre del cliente: Sancho
Informacion de sus cuentas
Cuenta de Ahorros        saldo: 2400.0
Cuenta de Cheques        saldo: 2000.0


Nombre del cliente: Rocinante
Informacion de sus cuentas
Cuenta de Ahorros        saldo: 3900.0
Cuenta de Cheques        saldo: 3000.0
```